CS152 FPGA CAD Tool Flow
University of California at Berkeley
College of Engineering
Department of Electrical Engineering and Computer Sciences

Compiled: 4/3/2003 for CS152 Spring 03, Prof. John Kubiatowicz
Updated: 9/4/2003 for CS152 Fall 03, Prof. Dave Patterson
Updated: 2/10/2004 for CS152 Spring 04, Prof. John Kubiatowicz
Sources: lab3.pdf from CS150 9/17/2002, Xilinx Reference Manual, Xilinx University Program
Please send any errors, corrections, or comments to cs152-staff@imail.eecs.berkeley.edu

1 Introduction to the Design Tool Flow
2 How to Use the Design Tool Flow
3 Basic Project Tutorial
4 ChipScope
5 Tips and Hints

1 Introduction to the Design Tool Flow

Refer to the illustration on the next page for the steps involved in the CAD tool flow we will use.

1.1 Design Entry

The first step in logic design is to conceptualize your design. Once you have a good idea about the function and structure of your circuit, you can start the implementation process by specifying your circuit. In this class we will use a Hardware Description Language (HDL) called Verilog. HDLs have several advantages over other methods of circuit specification: ease of editing (files can be written using any text editor), ease of management when dealing with large designs, and the ability to use a high-level behavioral description of a circuit. If you are familiar with emacs, you may find it convenient for writing and editing Verilog code. Alternatively, you may use the editor provided in Xilinx's Project Navigator or ModelSim.

1.2 Synthesis

Once your design is entered, the next step in the implementation path is synthesis. In our case, the function of the synthesis program is to translate the Verilog description of the circuit into an equivalent circuit comprising a set of primitive circuit components that can be directly implemented on an FPGA. In a way, the synthesis tool is almost like a compiler. Where a compiler translates to a sequence of primitive commands that can be directly executed on a processor, synthesis translates to primitive circuit components that can be directly implemented in FPGA. The final product of a synthesis tool is a netlist file, a text file that contains a list of all the instances of primitive components in the translated circuit and a description of how they are interconnected. The synthesis tool we will be using in this class is called Synplify. Note that Xilinx's software suite can also perform this synthesis procedure. The reason we use Synplify is because Synplify is an industrial strength CAD tool program. It's faster, produces better logic, and will give many more synthesis warnings—something very useful for students!

1.3 Placement, Routing

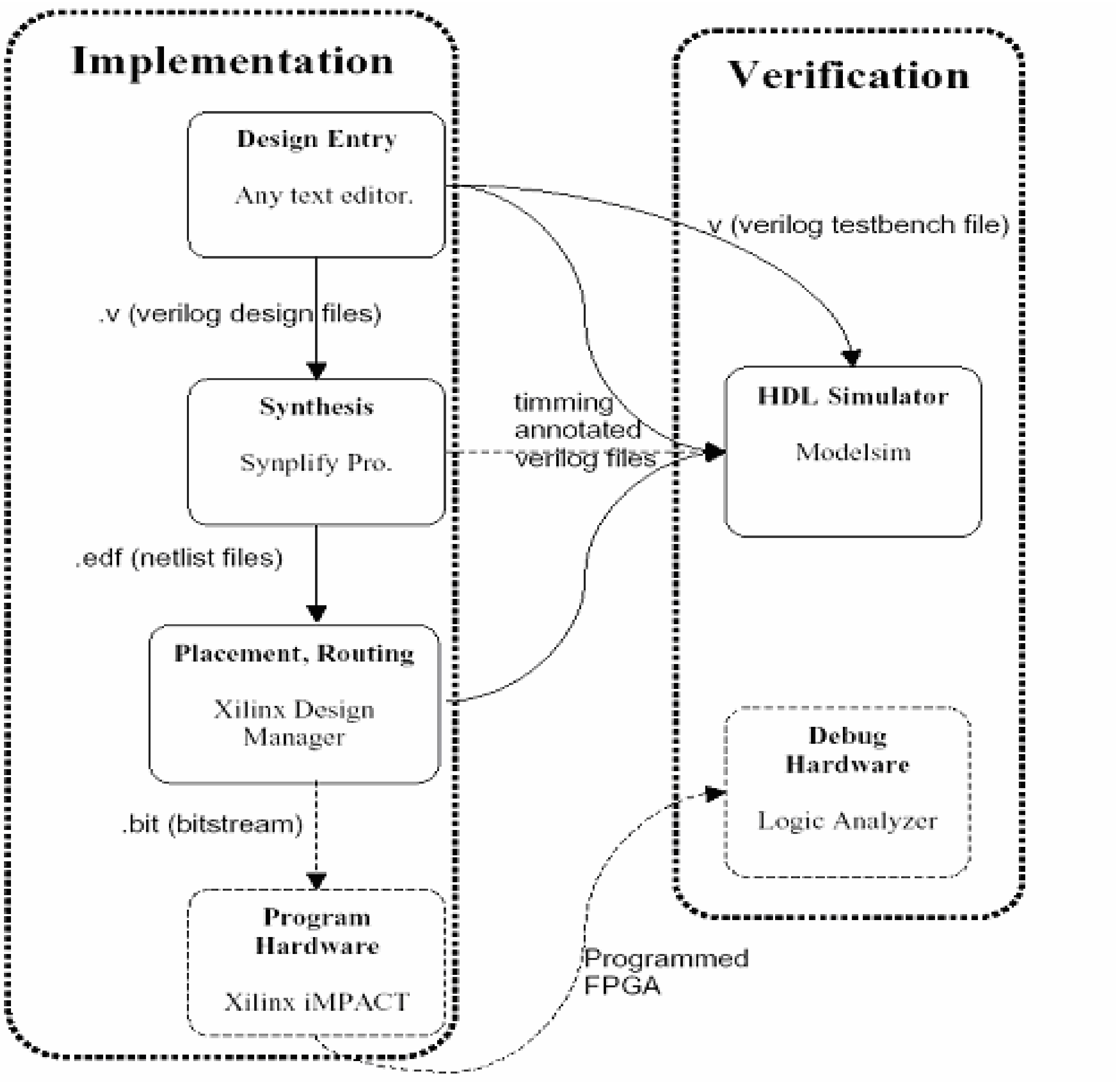
The next step in the implementation flow is to take the netlist of components generated by the synthesis tool and turn it into bits that are need to configure the LUTs, muxes, Flipflops, and other configurable resourses in the FPGA. To do that, first the primitive circuit components in the netlist need to be assigned to a specific *place* on the FPGA. For example, a 4LUT implementing the function of a 4 input NAND gate in a netlist could be implemented with any of the about 40,000 4LUTs in a Xilinx Virtex 2000E FPGA chip. Clever choice of placement will make the subsequent routing easier and result in a faster overall circuit. Once the components are placed, the proper connections must be made according to the netlist description obtained from the synthesis step. That step is called *routing*. Unlike synthesis, which only requires a set of primitive components to translate to; placement and routing are dependent upon the specific size and structure of the target FPGA chip. Due to this reliance, the FPGA vendor usually provides the placement and routing programs. Therefore, we will be using the Xilinx's Project Navigator to perform this step. The end product after placement and routing is a bit file containing the stream of bits used to configure the FPGA.

1.4 Program Hardware

The last step in the implementation flow is the simple act of transporting the configuration bits to the FPGA. There are also many ways of doing this. For this class we will be mostly using the Parallel Cable IV along with the iMPACT software to program the board.

1.5 Verification

As you should have learned from experience, a significant part of the time and effort spent on any sizable project will be spent on debugging, and logic design is no exception. There are two ways to verify the correctness of a design: to program the FPGA with the design and check if the circuit is behaving correctly, or to run simulations of the design in software. While programming the FPGA and physically checking the functionality sounds simple, the whole tool flow requires a significant amount of time to complete, especially as your designs get larger and more complex (20 minutes towards the end of the semester!). Repeatedly tweaking the input design to fix errors would require running the flow repeatedly, a huge waste of time. In addition it can be difficult to physically observe the causes for an error on a FPGA. For these reasons, software simulation is essential in the verification process. There are many places along the tool flow where you can use simulation to verify the correctness of your design. As you progress down the tool flow and more information about the physical implementation on the FPGA becomes available, more accurate timing simulations can be performed. **You should not attempt to verify that your design works on the board until it works in simulation!!!** To do so otherwise would just be a waste of time. In this class, we will be using ModelSim to do our verification.



CAD tool flow. Optional links are showed as dashed lines.

2 How to Use the Design Tool Flow – An Overview

2.1 Design Entry

Design entry will be done either through Verilog or Schematics using Xilinx ISE 5.2i. You can find a short tutorial below. Note that the Synthesis tool Synplify Pro does not synthesize schematic files, so you have to take the functional Verilog code of the schematics instead, so that you can synthesize your project. For the Xilinx primitives in the Verilog code you use, you may be required to created blackboxes for those primitive blocks as well.

To port your schematics into Verilog, follow the instructions in section 3 of this tutorial.

Creating blackboxes: You will need to create a blackbox for each specific Xilinx primitive block that is used in the functional Verilog code. A Xilinx primitive block is one that is predefined in the Xilinx block libraries: for example IBUF, AND3 and RAMB4_S16. To create a blackbox, all

you need to do is instantiate a module with the primitive name with the correct parameter signature and comment /* synthesis syn_black_box */ after the signature.
An example of a blackbox for the primitive IBUFG is:

```
/*****************************************************
 * IBUFG is a xilinx primitive for global clock buffering
 */
module IBUFG (I, O) /* synthesis syn_black_box */;
   input I;
   output O;
   //Behavior module of BUFG for simulation
   assign O = I;
endmodule
```

You will only need to create a black box if Synplify complains that it cannot find the primitive.

2.2 Functional simulation
Simulation of your design will be done mainly through ModelSim. You can also find a short tutorial below. After you have entered your design and have verified its functionality, you can move to synthesis and place and route.

2.3 Synthesis
Now we start to map the design to an FPGA:
1. Start the synthesis program Synplify Pro.
2. Start a new project from **File->New**, project file.
3. Add your Verilog source files to the project. Be sure to include the top_level file. Do not include your testbenches.
4. Change target device in the **implementation option** to Xilinx Virtex-E XCV2000E FG680. Speed Grade is –6.
5. Indicate the TopLevel.v as the top level module.
6. Click **RUN**
7. Synplify will create an EDIF file. The EDIF file is your netlist.

2.4 PlaceRoute
1.Start the Xilinx Design manager by choosing from Synplify Pro, **Options->Xilinx->Start Project Navigator.** Note: Synplify Pro will automatically create a project for you this way, using the EDIF file. However, the version created will be for an older version of Xilinx's Project Navigator. Xilinx will ask you if you want to update the project to the current version. Choose **Yes** to update, and select **No** for archiving. Unfortunately, this will cause your target device to be changed. Under Sources in Project, it should say "xcv2000e-6fg680 – EDIF." If it does not, right click on it and go to properties, and adjust the settings.
2. Under Processes for Current Source, right click on **Generate Programming File** and bring up the properties window. On the **Readback Options** tab make sure that the following checkboxes are checked: Create **Readback Datafiles** and **Create Mask File**. If these two options are not set then you will not be able to program using Boundary-Scan mode (and therefore you will not be able to use ChipScope).
3. Under Processes for Current Source, click on **Generate Programming File.**

2.4b – Alternative Way
1. If you do not want to go through the trouble of updating your project, you may simply use project navigator and create a new project, with the following settings:

Device Family: VirtexE
Device: xcv200e
Package: fg680
Speed Grae: -6
Design Flow: EDIF
2. Click **OK** to create the project.
3. Go to **Project -> Add Copy of Source**. Find the .edf file created by Synplify and add it to your project.
4. Under Processes for Current Source, right click on **Generate Programming File** and bring up the properties window. On the **Readback Options** tab make sure that the following checkboxes are checked: Create **Readback Datafiles** and **Create Mask File**. If these two options are not set then you will not be able to program using Boundary-Scan mode (and therefore you will not be able to use ChipScope).
5. Under Processes for Current Source, click on **Generate Programming File.**

2.5 Download Design to the Board
1. Make sure that the parallel cable is connected to the board and that it is connected to the **JTAG** port on the board **NOT** the **Slave Serial** port.
2. start iMPACT: In the **Processes for Current Source** window, double click **Configure Device (iMPACT)** selection at the very bottom of the window. The previous step should have already created a .bit file to be programmed. This step should open up iMPACT, the tool actually used to program the board.
3. Connect cable: iMPACT should start up and ask you what operation mode to be in. Select **Configure Devices.** When it asks how you want to configure the device, select **Boundary-Scan Mode**. It will next ask you whether or not you want automatically connect or to enter a boundary scan chain, select **Automatically connect to cable and identify Boundary-Scan chain**. Depending upon how the computer is setup it will find one or two devices. One will be labeled **xccace** (depending upon the configuration of the computer this device may not be present) and the other will be labeled **xcv2000e**. If the connection fails, the most likely causes are that you have more than one iMPACTs open, the board is off, or you have not connected the parallel cable to the right port on the board.
4. Loading the bit file: next the program will prompt you to select a bit file for each of the devices that it found. If **xccace** was found then click the bypass button in the **Assign New Configuration File** dialog box. Once you are programming the **xcv2000e** device (the board) select your bitfile from the window. Make sure you select the one in the right folder, as iMPACT may not always open up in your expected folder! You can also double click the chip picture and manually load the bit file for the source module you want from your project directory.
5. Download to board: Right click on the chip picture labeled **xcv2000e**, and choose **Program** If the program was downloaded to the board, you should see a “Programming Successful” message. If it does not work, most likely the board is not connected correctly and you should make sure the parallel cable is connect to the JTAG slot and also that the board is turned on.

That’s it!

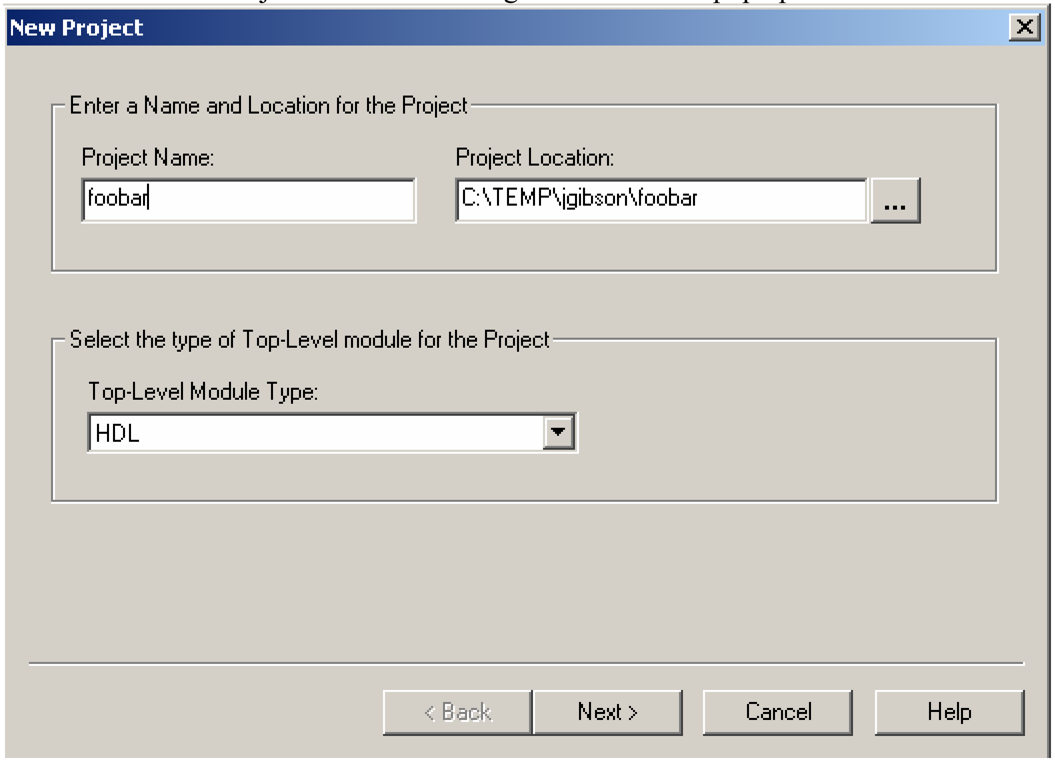
The rest of the tutorial shows how to use the Xilinx Project Navigator to create a project, and enter your design. It also covers how to use the schematics editor, and how to use Modelsim.

3 Basic Project Tutorial

3.1 Creating a Project
3.2 Adding New Modules
i. Adding a Verilog Source
ii. Adding a Schematic Source
iii. To view your schematic in Verilog HDL
iv. Creating a Schematic Symbol from Verilog or Schematic Sources
v. Combining Verilog and Schematic Sources
3.3 Simulating your design
i. Adding Waves

3.1 CREATING A PROJECT

1) Load up Xilinx Project Navigator.
2) Go to File→ New Project and the following window should pop up:


New Project

Enter a Name and Location for the Project

Project Name: foobar

Project Location: C:\TEMP\jgibson\foobar

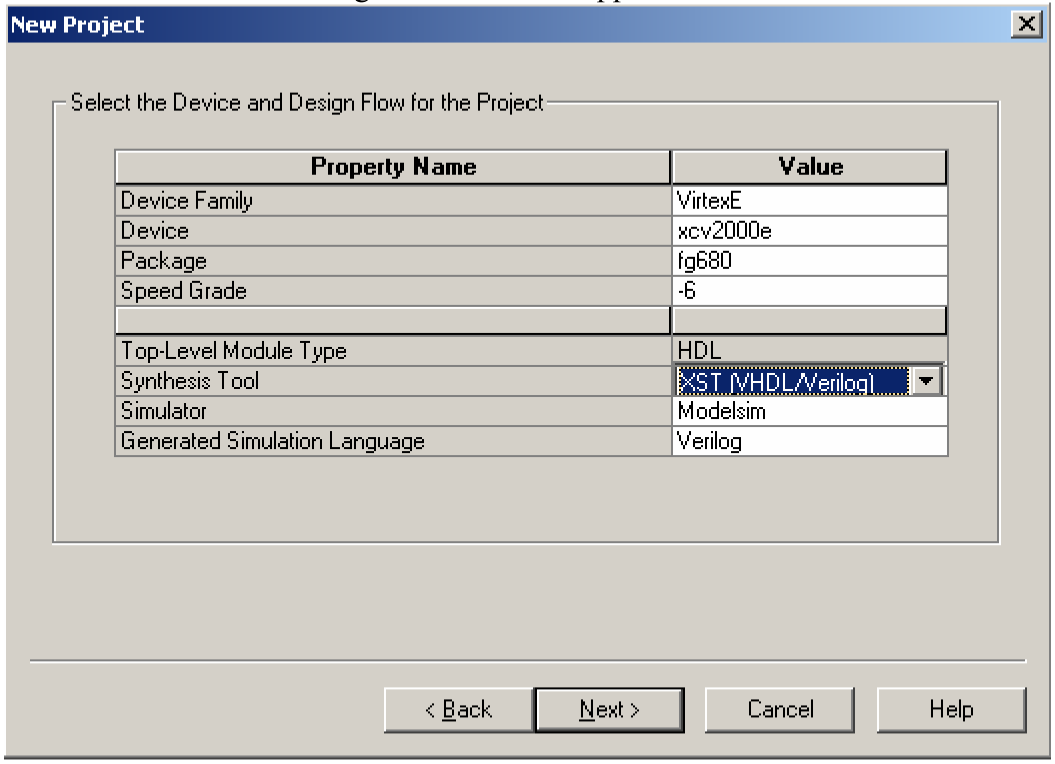
Select the type of Top-Level module for the Project

Top-Level Module Type: HDL

3) Make the project settings the same as the ones seen above.

Project Name	What you wish to call the project
Project Location	Place in c:\Temp and later move the project directory into your U:\ for storage
Top-Level Module Type	HDL

4) Click next and the following window should appear



Property Name	Value
Device Family	VirtexE
Device	xcv2000e
Package	fg680
Speed Grade	-6
Top-Level Module Type	HDL
Synthesis Tool	XST (VHDL/Verilog)
Simulator	Modelsim
Generated Simulation Language	Verilog

5) Set the project settings as seen above and click next when you're finished:

Device Family	VirtexE
Device	xcv2000e
Package	fg680
Speed Grade	-6
Synthesis Tool	XST (Verilog/VHDL)
Simulator	Modelsim
Generated Simulation Language	Verilog

6) Now you'll have the opportunity to add a new source to the project, you can do this now or later. Hit next when you're finished.
7) Now you can add existing sources to the project. Like adding a new source this can be done after the project is created as well. Hit next when you're finished.

You should be on the final screen of the New Project wizard. It will list the settings that you have chosen, verify that they are correct and then click Finish to create the project.

3.2 ADDING NEW MODULES

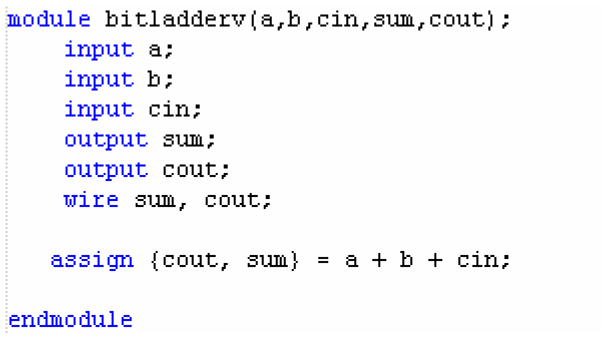
There are 3 types of files you will add: Verilog Modules, Schematics and Verilog Test Fixtures.

Adding Verilog Modules:

1) Go to Project→New Source and then Select Verilog Module. Under *File Name*, give the Verilog module you want to create an appropriate name. * **Note:** names should start with an alphabetic letter (meaning do NOT begin the name with a digit or special character). For this tutorial, call the module "bit1adderv". Click Next.

2) The next window that pops up allows you to specify the inputs and outputs of your Verilog module. You don't have to specify them all at the beginning; you can add more inputs and outputs directly in the Verilog file. You can add inputs a, b, cin and outputs sum and cout now, but it's usually easier to add them directly in the Verilog file later. Click next. And then click finish.

3) The Verilog file should pop up and edit it from there. You can find details on how to program in Verilog elsewhere. Don't forget to save when you're done.
Your code for bit1adderv.v should look somewhat like this:



```
module bit1adderv(a,b,cin,sum,cout);
    input a;
    input b;
    input cin;
    output sum;
    output cout;
    wire sum, cout;

   assign {cout, sum} = a + b + cin;

endmodule
```

Adding Schematic Modules:

Now you will learn how to make schematics, which basically means you will be creating logic designs down to each single gate and wire.

1) Go to Project→New Source and then Select Schematic Module. Under *File Name*, give the Schematic module you want to create an appropriate name. *** Note:** names should start with an alphabetic letter (meaning do NOT begin the name with a digit or special character). For this tutorial, call the module "bit1adder". Click Next. And then click finish.

2) ECS, Xilinx's schematic program should load. A blank sheet should appear.

To add basic **logic gates** click on the button that looks like the following:

Or you can use the hot-key Ctrl-F
Two windows on the right hand side should appear including all the gates that you can add to the schematic. It should be fairly self explanatory. Press the ESC key when you are done adding symbols/gates.

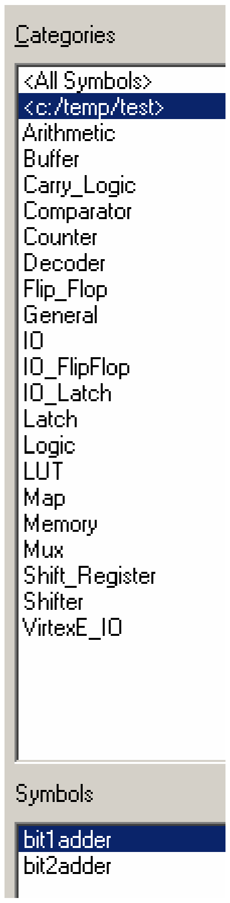
*Note:

Schematic symbols you create within your project can be accessed by selecting the directory in which you placed your project.

Go back to project navigator. Select the bit1adder.v file, and then double click on the *Create Schematic Symbol* under *Processes for Current Source* inside of *Design Entry Utilities.*

This will allow you to use the block you created in the source file in the ECS schematic editor program. This allows you to reuse a block you created in ECS without having to redraw it over and over again.

In the picture on the right, the project was placed in c:/temp/test and two symbols created for that project from Verilog or schematic sources are named "bit1adder" and "bit2adder".



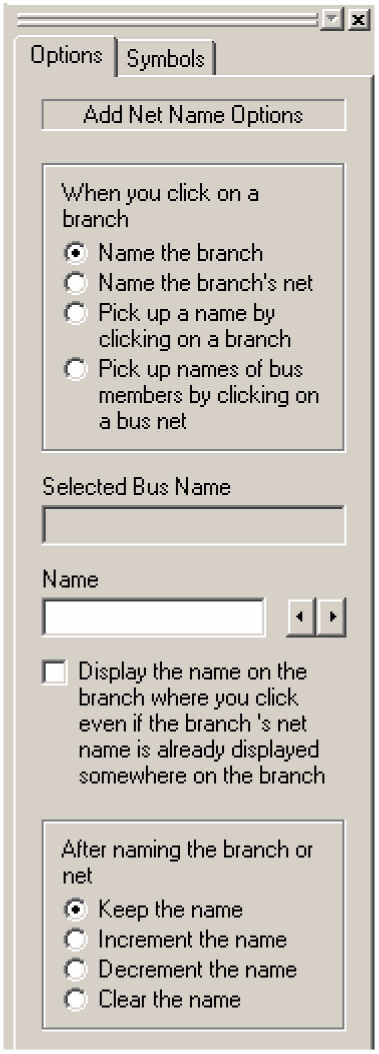
To add **wires** click on the button that looks like the following:

Or you can use the hot-key Ctrl-W
Put your mouse head over the wire you want to begin the wire until you see a red outline and left click. Then drag your mouse and the wire to the net you want to connect it too until you see a red outline and left click. The wire then should be connected. Press the ESC key when you are done. Use can use the same method to add busses.

To **name nets/wires** click on the button that looks like the following:

Or you can use the hot-key Ctrl-D
The following window shows up on the left hand side of the screen:

Type the net name in the middle text box:



Options | Symbols

Add Net Name Options

When you click on a branch

- [✓ Name the branch]
- Name the branch's net
- Pick up a name by clicking on a branch
- Pick up names of bus members by clicking on a bus net

Selected Bus Name

Name

- Display the name on the branch where you click even if the branch 's net name is already displayed somewhere on the branch

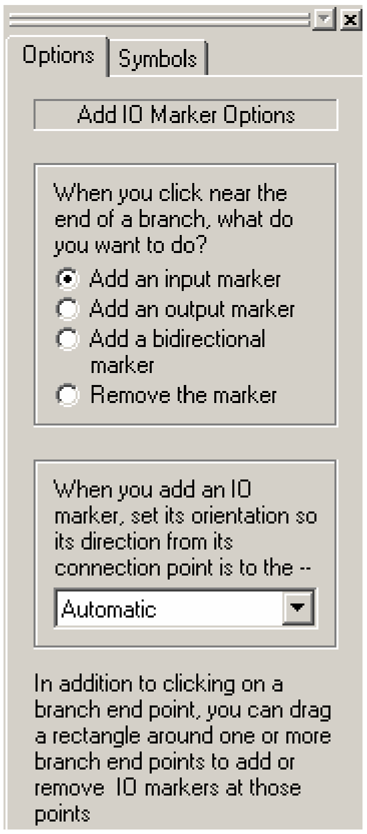
After naming the branch or net

- [✓ Keep the name]
- Increment the name
- Decrement the name
- Clear the name

Finally you should see the net name at the end of your cursor when you place the cursor over the schematic window. Place the cursor over the wire/net you wish to name and left click. Press the ESC key when you are done. Note that you can connect pins by giving them the same name, or by physically connecting a wire.

To add **I/O pins** click on the button that looks like the following:

Or you can use the hot-key Ctrl-G
Select the type of pin you wish to add in the box as seen below:



Options | Symbols

Add IO Marker Options

When you click near the end of a branch, what do you want to do?

- [✓ Add an input marker]
- Add an output marker
- Add a bidirectional marker
- Remove the marker

When you add an IO marker, set its orientation so its direction from its connection point is to the --

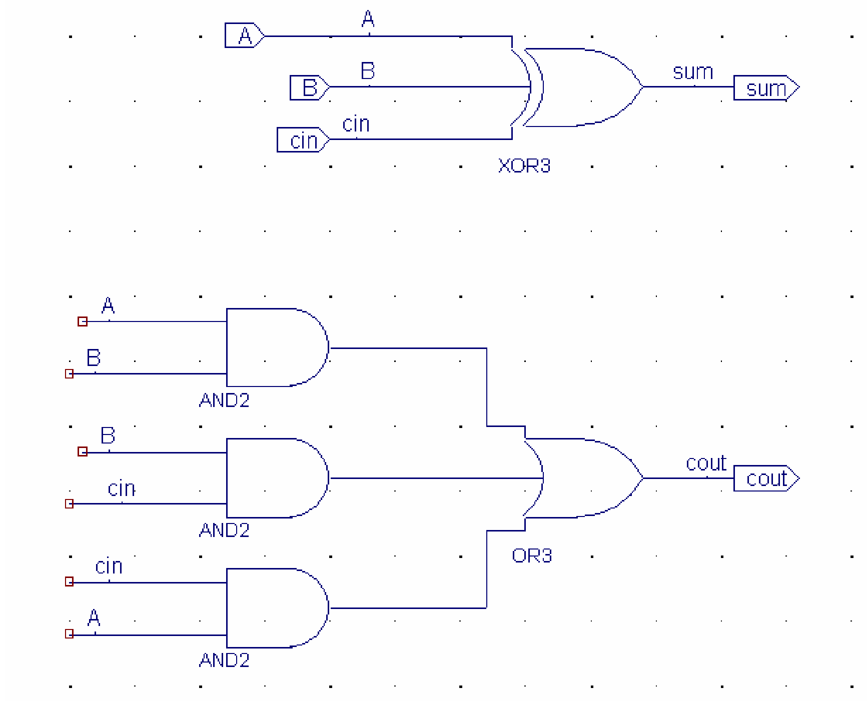
Automatic

In addition to clicking on a branch end point, you can drag a rectangle around one or more branch end points to add or remove IO markers at those points

Finally place the cursor over the wire/net you wish to add the pin to and left click. Press the ESC key when you are done.

This should be enough for you to complete a basic design. If you need more help consult the Xilinx ECS manual.

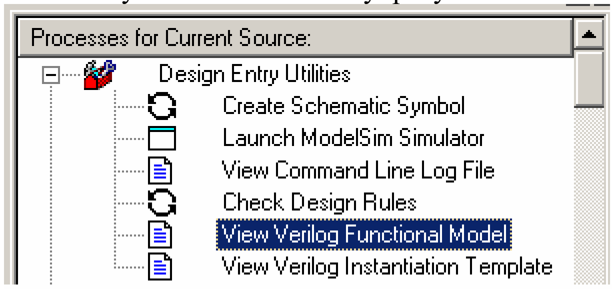
Now after playing around with the tools, create a 1 bit adder that looks like the following:



Don't forget to save when you're done.
The schematic file "bit1adder.sch" should then appear in the *Sources in Project* window in Xilinx Project Navigator.

To **view your schematic in Verilog** HDL

Select your schematic file in the *Sources in Project* window. Then in the *Processes for Current Source* window, expand the Design Entry Utilities box. Double click View Verilog Functional Model and the Verilog file should open. The file is read-only though. You can go to *File -> Save As*, and save it as a .v file so that you can add it into Synplify later.


Processes for Current Source:
Design Entry Utilities
Create Schematic Symbol
Launch ModelSim Simulator
View Command Line Log File
Check Design Rules
View Verilog Functional Model
View Verilog Instantiation Template

Combining Verilog and Schematics

You can easily combine schematic and Verilog modules in your project. For this tutorial, we will use both 1 bit adders you created in Verilog and in schematic by "wrapping" both adders in a Verilog file to create a 2 bit adder. Basically in Verilog you can reference each schematic module by just referencing the name of the schematic block.

1. Create a new Verilog module as detailed in the previous sections named "bit2adder".
2. For the appropriate inputs and outputs, don't forget some are 2 bit busses.
3. Your code should look somewhat like the following:

```
module bit2adder(a,b,cin,sum,cout);
    input [1:0] a;
    input [1:0] b;
    input cin;
    output [1:0] sum;
    output cout;
    wire ctemp;
   // note that the input output position of both adders are different
   // cout precedes sum out in bitladder though the opposite occurs for bitladderv
   bitladder ba (a[0], b[0], cin, ctemp, sum[0]);
   bitladderv bav (a[1], b[1], ctemp, sum[1], cout);

endmodule
```

4. Don't forget to save the file.

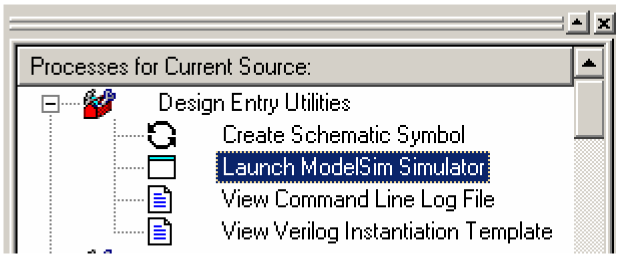
3.3 SIMULATING YOUR DESIGN

In this course, you can also use ModelSim to simulate your designs. It allows you to run testbenches and it can generate waveforms for you to view your design's outputs. We will only be covering one of many ways to use ModelSim below:

Simulating your design using a testbench module.
First, we need to create a testbench. In *Project Navigator*, add a new Verilog file called bit2adder_tb.v. It should look something like this:

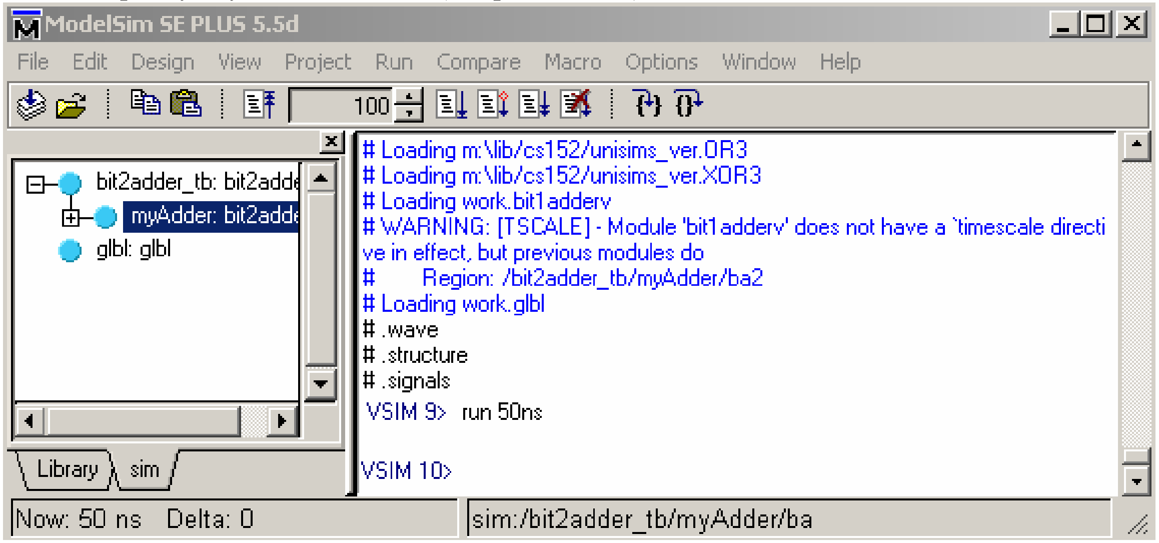
```
`timescale 1ns/1ps

//note the backwards apostrophe before the t in
//timescale--it's the button to the left of the number 1 on the keyboard
//1ns refers to how fast time goes in this file. i.e. #2 means 2 nanoseconds
//1ps refers to the resolution at which things are calculated

module bit2adder_tb();

     reg [1:0] a, b;
     reg cin;
     //inputs to the module you're testing should be regs
     //this way, you can actually assign them a value

     wire [1:0] sum;
     wire cout;
     //outputs to the module you're testing should be wires

     bit2adder myAdder(.a(a),
                       .b(b),
                       .cin(cin),
                       .sum(sum),
                       .cout(cout));

     initial begin
          a = 2'b00;
          b = 2'b00;
          cin = 1'b0;
          #10
          a = 2'b01;
          #10
          b = 2'b10;
          #10
          cin = 1'b1;
     end
endmodule
```

Now, we can begin testing. First select your previously created testbench source in the *Sources in Project* window.


Processes for Current Source:
Design Entry Utilities
Create Schematic Symbol
Launch ModelSim Simulator
View Command Line Log File
View Verilog Instantiation Template

In the *Processes for Current Source* window, expand the *Design Entry Utilitie* , and double click *Launch ModelSim Simulator.*

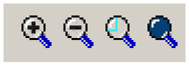
This will launch ModelSim which will load your testbench (simulate the input values you set in the testbench). You then need to tell ModelSim to actually run your code, by typing in *run 40ns*. You can specify any amount of time (use ps, ns, or us).



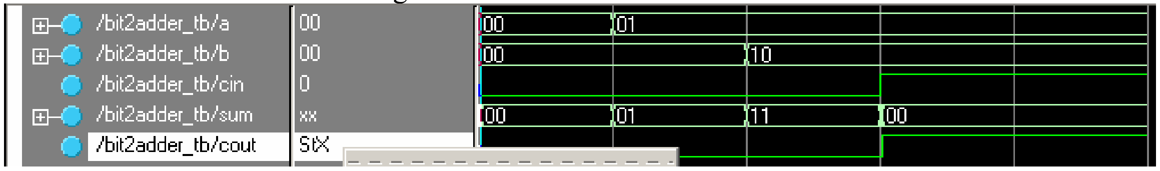
```
# Loading m:\lib/cs152/unisims_ver.OR3
# Loading m:\lib/cs152/unisims_ver.XOR3
# Loading work.bit1adderv
# WARNING: [TSCALE] - Module 'bit1adderv' does not have a `timescale directi
ve in effect, but previous modules do
#       Region: /bit2adder_tb/myAdder/ba2
# Loading work.glbl
# .wave
# .structure
# .signals
 VSIM 9>  run 50ns

VSIM 10>
```

Now: 50 ns   Delta: 0   sim:/bit2adder_tb/myAdder/ba

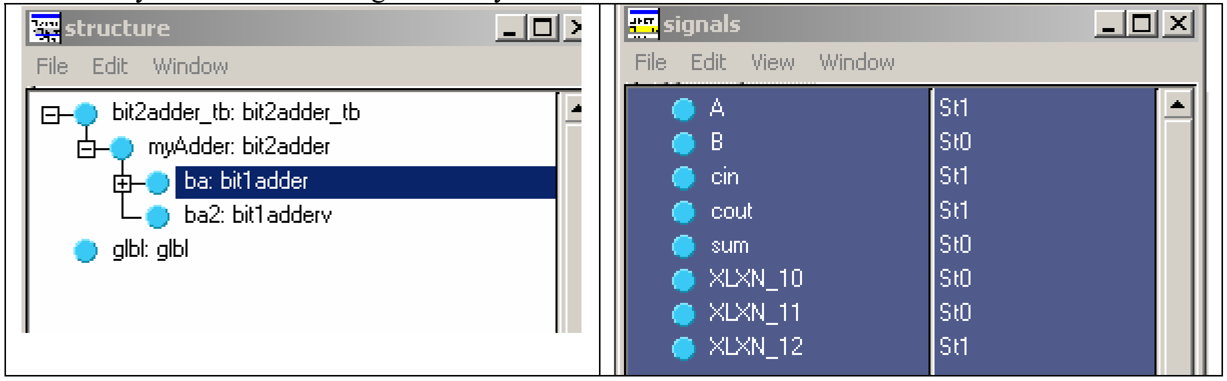
The resulting waveforms in the wave window may not be easily seen b/c the time interval between clock cycles may be too long or too short. To remedy this you can either use the zoom out tool (the magnifying glass with the plus sign) to shorten the length between clock cycles and view more of the wave at once, or the zoom in tool to do the opposite. You can also click on the very right magnifying class to zoom full.



Your wave should look something like this:



Note that you can add more signals into your waves!

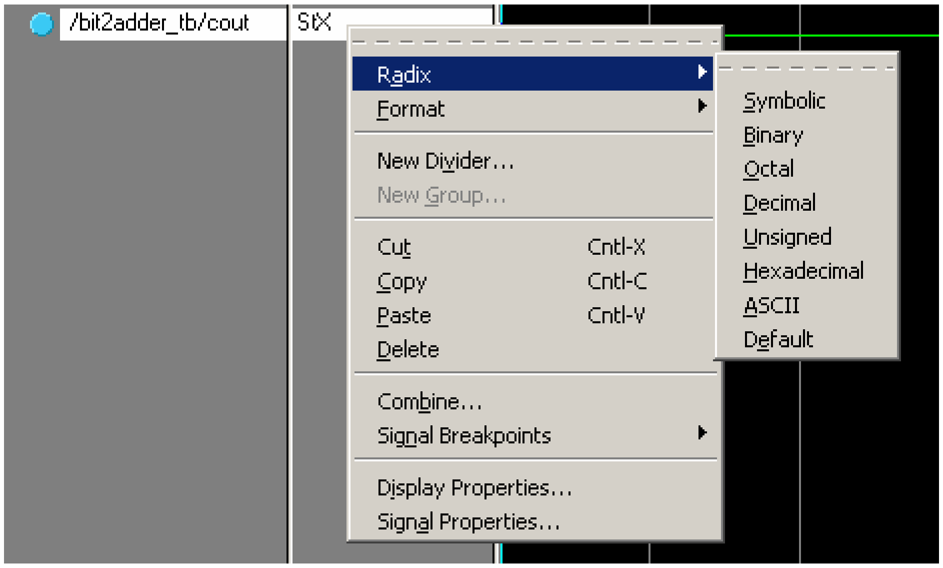


A	St1
B	St0
cin	St1
cout	St1
sum	St0
XLXN_10	St0
XLXN_11	St0
XLXN_12	St1

Under the structure box, you need to select where you want your signals to be coming from. In the above example, we want the signals internal to the ba instantiation of the bit1adder. The signals box on the right contain the actual signals themselves. Click and drag the ones you want into the wave diagram.

Now you have the ability to write a testbench which will allow you to run your design, and see what each and every signal is at any moment in time. This is a very powerful debugging tool, and is one that you will put to heavy use in the weeks ahead.

A final note: If you right click on a signal inside the wave diagram, a box with plenty of options opens up. For example, you can choose to display the signal's value as binary, hex, or unsigned numbers, among others. Use these options to aid you in your debugging. We encourage you to play around with Modelsim, as there are many, many, more features that are not discussed here.



4. ChipScope

ChipScope is a software logic analyzer that interfaces directly with the FPGA to give you data on your designs. Note that ChipScope uses up resources on the board, so once your design is fully debugged you should disable the ChipScope module for better performance.

There are two ways to insert ChipScope modules into your design. The first is to use the ChipScope Pro Core Generator with which you will generate verilog modules to place directly in your design. The second is to use the ChipScope Pro Core Inserter to connect the ChipScope cores to wires found in your EDN file after synthesizing your design. This tutorial will cover ChipScope Pro Core Generator, but feel free to experiment with the Core Inserter.

Generating the Cores:

1. First start **ChipScope Pro Core Generator**.
2. You will be presented with a list of cores to generate, select **ICON (Integrated Controller)**.

3. Next you should select the directory into which the generated cores will be placed. A good place for them is a subdirectory of your project directory.
4. Make sure that the **Device Family** is set to **VirtexE**.
5. There are several other options including the number of control ports. This tutorial will only cover the more basic options available. So just leave the number of control ports set to one. Tweak these settings as necessary and then proceed to the next screen.
6. Make sure that the **HDL Language** setting is **Verilog** and that the **Synthesis Tool** is **Synplicity Synplify**. Then click Generate Core to generate the core.

The next step is to generate a logic analyzer to do the actual work on the chip.

1. **Start ChipScope Pro Core Generator** if it is not already running.
2. Select **ILA (Integrated Logic Analyzer)** from the list of cores to generate.
3. Make sure that the directory to place the cores into is selected properly.
4. Make sure that the **Device Family** is set to **VirtexE**. Advance to the next screen.
5. Now select the number of trigger ports to use as well as their properties. For this example the default settings will be used (1 port with a width of 8 bits, 1 match unit, and basic matching). Advance to the next screen when you are finished.
6. You should be on the Data Options screen. Here you select the depth of the sampling as well as the width of the data port to use. Deeper sampling depths and wider data widths will increase the number of block rams that are consumed on the board. If you do not need many data lines to inspect then you can check the **Data Same As Trigger** checkbox to disable the additional data ports. When you are satisfied with the settings then you can advance to the next screen.
7. Make sure that the **HDL Language** setting is **Verilog** and that the **Synthesis Tool** is **Synplicity Synplify**. Then click Generate Core to generate the core.

Adding the Cores to your Design:

1. First add the verilog example modules to your design. They should be found in the directory where you placed your cores. Their names will probably be **icon_synplicity_example.v** and **ila_synplicity_example.v**.
2. Look inside these files and you will find a black box definition of each module along with example modules containing instantiations of each module. Make sure that the definitions for each module have the following synthesis directives after them otherwise Synplify will optimize them away: /* synthesis syn_black_box syn_noprune=1 */
3. Place an instantiation of each module in a file with the signals that you want to debug. Connect the control ports of the icon and the ila with a wire. Next give the ILA the clock signal that you want it to sample data upon. Finally pick which wires you want to use for triggers and data and feed them into the ILA.

Implementing the Design:

After you have completed synthesizing the design you must place the *.edn files into the same directory as the .edn file that Synplify generated. Otherwise when you try to Translate / Map / Place and Route you will get errors about the icon and ila black boxes not being found.

Actually using the Logic Analyzer:

Once your design is on the board and running you can actually use ChipScope to debug your design.

1. Start the **ChipScope Pro Analyzer**. You can run it either run it directly from the Start Menu or you can use the **Analyze Design Using ChipScope** option from Processes window in Project Navigator.
2. Under the **JTAG Chain** menu select **Xilinx Parallel Cable** let it auto detect the cable type. Leave the speed at 2.5 MHz, if you get strange errors when trying to use this then you should try reducing the speed until it functions properly. Advance to the next screen, and click OK.
3. Now you should see your ILA listed in a hierarchy in the upper left corner. Select the **Trigger Setup** item under your ILA unit.
4. Now in the **Trigger Setup** menu at the top of the screen select the **Trigger Immediate** option. This should cause the waveform window to fill up with data.
5. You can also set triggers to capture data for you at more interesting times. In the trigger setup window select the value setting of your match unit. Now type in a combination of values that will be triggered by your trigger port (X's are don't cares, 1 is high, and 0 is low). If you didn't connect your trigger ports to anything useful then this functionality will be useless to you.

There are many other options that were not explored in this tutorial, feel free to examine them yourself.

5. Tips and Hints

These may not all make sense now, but read them and refer back to them as needed.

- Do all your work on the local C:\TEMP directory. It will be much faster. Don't forget to save your work back to your U:\ after you finish!
  This is even more important in 125 Cory where anything on the C drive that you create WILL BE AUTOMATICALLY DELETED when you logout.
- When uploading the bitstream to the board, check the timestamp of the bitstream. Often times, you will load old bitstreams due to the fact that they're all named the same, in similar folders.
- Do not name any parameters in Verilog as "init." It will not place and route and the CAD tools will not tell you why. TRUST ME on this—I learned this lesson at a cost or 12 hours.
- Synplify will not synthesize Schematics. All schematics must be ported into a Verilog file first.
- Work together! You are a community, and will be spending <u>a lot</u> of time together. Be friendly and helpful—you never know when you'll need somebody else's help.
- If you have some modules or signals that you want to include only in synthesis, like your ChipScope cores then you can wrap them in directives like so:

```
`ifdef synthesis

wire [35:0] control;

icon myicon (.control0(control));
```

```
ila myila
(.control(control), .clk(clk), .data(foobar), .trig0(baz));

`endif
```

- If you have two versions of a module one for synthesis and the other for simulation you can do something like this:

```
`ifdef synthesis

synonly example (.in(foo), .out(bar));

`else

simonly example (.in(foo), .out(bar));

`endif
```

- However if you have something that you only want in simulation DON'T DO THIS:

```
`ifdef synthesis
`else

simonly example (.in(foo), .out(bar));

`endif
```

The tools (at least previous versions of them) will get confused and not work properly. Instead do this:

```
// synthesis translate_off

simonly example (.in(foo), .out(bar));

// synthesis translate_on
```